Applied Informatics C++ Coding Style Guide

Rules and Recommendations

Version 1.5

Purpose of This Document

This document describes the C++ coding style employed by Applied Informatics.

The document is targeted at developers contributing C++ source code to the products of Applied Informatics, including contributions to open-source projects like the POCO C++ Libraries.

Copyright, Licensing, Trademarks, Disclaimer

Acknowledgements

This work is based on various guidelines found on the internet, especially the rules and recommendations by Mats Henricson and Erik Nyquist. Also incorporated are ideas and recommendations from various other sources. See the Appendix for a list of recommended resources.

Table of Contents

1 Introduction 5

2 Terminology 6

3 General Recommendations 9

4 Source Files and Project Structure 10

4.1 Structure of Source Code **10**
4.2 Naming Files **10**
4.3 Project Structure **10**
4.4 Whitespace **11**
4.5 Comments **12**
4.6 Header Files **13**

5 Names 16

6 Style 19

6.1 Classes **19**
6.2 Functions **20**
6.3 Templates **21**
6.4 Compound Statements **21**
6.5 Flow Control Statements **21**
6.6 Pointers and References **23**
6.7 Miscellaneous **23**

7 Classes 24

7.1 Access Rights **24**
7.2 Inline Functions **24**
7.3 Friends **25**
7.4 Const Member Functions **26**
7.5 Constructors and Destructors **26**
7.6 Assignment Operator **29**
7.7 Operator Overloading **30**
7.8 Member Function Return Types **30**
7.9 Inheritance **31**

8 Class Templates 32

9 Functions 33

9.1 Function Arguments **33**
9.2 Function Overloading **34**
9.3 Formal Arguments **34**
9.4 Return Types and Values **34**

9.5 Inline Functions 35
9.6 Temporary Objects 35
9.7 General 36

10 Constants 37

11 Variables 38

12 Pointers and References 39

13 Type Conversions 40

14 Flow Control 41

15 Expressions 43

16 Memory and Resources 44

17 Types and Namespaces 46

18 Error Handling 48

19 Portability 49

20 References and Recommended Reading 50

21 Appendix: Documentation 52
21.1 General Conventions 52
21.2 Documenting Classes and Structs 52
21.3 Documenting Functions 53
21.4 Documenting Enumerations 54
21.5 Documenting Types 54
21.6 Libraries, Packages and Modules 54

22 Appendix: Abbreviations 55

1 Introduction

The purpose of this document is to define one style of programming in C++. The rules and recommendations presented here are not final, but should serve as a basis for continued work with C++. This collection of rules should be seen as a dynamic document; suggestions for improvements are encouraged. Suggestions can be made via e-mail to poco@appinf.com.

Programs that are developed according to these rules and recommendations should be:

- correct
- easy to maintain.

In order to reach these goals, the programs should:

- have a consistent style,
- be easy to read and understand,
- be portable to other architectures,
- be free of common types of errors,
- be maintainable by different programmers.

Questions of design, such as how to design a class or a class hierarchy, are beyond the scope of this document. Recommended books on these subjects are indicated in the appendix entitled References.

In order to obtain insight into how to effectively deal with the most difficult aspects of C++, the provided example code should be carefully studied. C++ is a difficult language and there may be a very fine line between a feature and a bug. This places a large responsibility upon the programmer, as this is always the case when using a powerful tool. In the same way as for C, C++ allows a programmer to write compact and, in some sense, unreadable code.

In order to make the code more compact, the examples provided do not always follow the rules. In such cases, the broken rule is clearly indicated.

2 Terminology

1. An **identifier** is a name used to refer to a variable, constant, function or type in C++. When necessary, an identifier may have an internal structure consisting of a prefix, a name, and a suffix (in that order).
2. A **class** is a user-defined data type consisting of data elements and functions operating on that data. In C++, this may be declared as a class; it may also be declared as a struct or a union. Variables defined in a class are called member variables and functions defined in a class are called member functions.
3. A class/struct/union is said to be an **abstract data type** if it does not have any public or protected member variables.
4. A **structure** is a user-defined consisting of public member variables only.
5. **Public members** of a class are member variables and member functions that are accessible from anywhere by specifying an instance of the class and the name.
6. **Protected members** of a class are member variables and member functions that are accessible by specifying the name within member functions of derived classes.
7. A **class template** defines a family of classes. A new class may be created from a class template by providing values for a number of arguments. These values may be names of types or constant expressions.
8. A **function template** defines a family of functions. A new function may be created from a function template by providing values for a number of arguments. These values may be names of types or constant expressions.
9. An **enumeration** type is an explicitly declared set of symbolic integral constants. In C++ it is declared as an `enum` or `enum class`.
10. A **typedef** is another name for a data type, specified in C++ using a `typedef` or `using` declaration.
11. A **reference** is another name for a given variable. In C++, the "address of" (`&`) operator is used immediately after the data type to indicate that the declared variable, constant, or function argument is a reference.
12. A **macro** is a name for a text string defined in a `#define` statement. When this name appears in source code, the compiler's preprocessor replaces it with the defined text string.
13. A **constructor** is a function that initializes an object.

14. A **copy constructor** is a constructor in which the only argument is a reference to an object that has the same type as the object to be initialized.
15. A **move constructor** is a constructor in which the only argument is a rvalue-reference to an object that has the same type as the object to be initialized.
16. A **default constructor** is a constructor with no arguments.
17. An **overloaded function** name is a name used for two or more functions or member functions having different argument types.
18. An **overridden member function** is a member function in a base class that is re-defined in a derived class. Such a member function is declared virtual.
19. A **pre-defined data type** is a type defined in the language itself, such as `bool` or `int`.
20. A **user-defined data type** is a type defined by a programmer in a `class`, `struct`, `union`, `enum`, or `typedef/using` definition or as an instantiation of a class template.
21. A **pure virtual function** is a member function for which no definition is provided. Pure virtual functions are specified in abstract base classes and must be defined (overridden) in derived classes.
22. An **accessor** is a function returning the value of a member variable.
23. A **mutator** is a function modifying the value of a member variable.
24. A **forwarding function** is a function doing nothing more than calling another function.
25. A **constant member function** is a function that must not modify data members.
26. An **exception** is a run-time program anomaly that is detected in a function or member function. Exception handling provides for the uniform management of exceptions. When an exception is detected, it is thrown (using a `throw` expression) to the exception handler.
27. A **catch clause** is code that is executed when an exception of a given type is raised. The definition of an exception handler begins with the keyword `catch`.
28. An **abstract base class** is a class from which no objects may be created; it is only used as a base class for the derivation of other classes. A class is abstract if it includes at least one member function that is declared as pure virtual.
29. An **iterator** is an object which, when invoked, returns the next object from a collection of objects.
30. The **scope** of a name refers to the context in which it is visible.

31. A **compilation unit** is the source code (after preprocessing) that is submitted to a compiler for compilation (including syntax checking).

3 General Recommendations

Recommendation 1

Optimize code only if you know for sure that you have a performance problem. Think twice before you begin. Then **measure** and think again. C++ compilers are pretty good at optimizing these days. So, source code that "looks fast" does not necessarily produce faster object code. It only is harder to understand and maintain.

As C.A.R. Hoare once said: *Premature optimization is the root of all evil.*

Recommendation 2

Always compile production code with at least a second compiler and on a second platform. If the code has been developed with Microsoft Visual C++ under Windows, compile and test the code with the GNU C++ or Clang C++ compiler on a Unix platform, and vice versa. Even better is testing with a third compiler. This brings to light subtle errors and portability problems.

Recommendation 3

Always compile at the highest warning level possible. A lot of bugs can be avoided by paying attention to compiler diagnostics.

Rule 1

If this style guide leaves anything unclear, see how it has been done in the existing code base and do it accordingly.

4 Source Files and Project Structure

4.1 Structure of Source Code

Rule 2

Header (include) files in C++ always have the file name extension ".h".

Rule 3

Implementation files in C++ always have the file name extension ".cpp".

4.2 Naming Files

Rule 4

Always give a file a name that is unique in as large a context as possible.

A header file for a class should have a file name of the form <class name> + extension. Use uppercase and lowercase letters in the same way as in the source code.

Since class names must generally be unique within a large context, it is appropriate to utilize this characteristic when naming its header file. This convention makes it easy to locate a class definition using a file-based tool.

4.3 Project Structure

Rule 5

Every project (library or application) has its own directory with a well-defined structure.

A project directory contains project files, make files, as well as other directories for header files, implementation files, documentation and the test suite.

Figure 1 shows the directory hierarchy for a project.

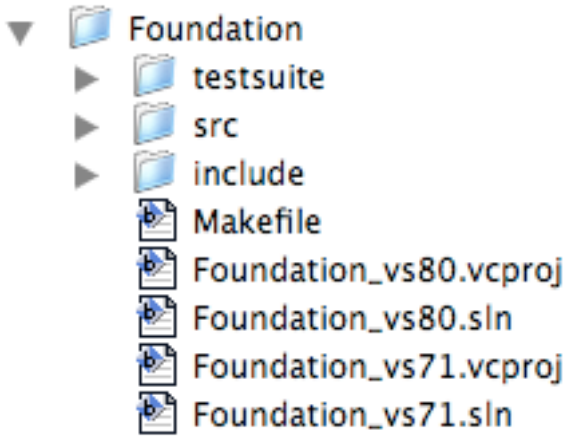

Figure 1: Project directory hierarchy

A test suite is a project in itself and thus has the same general structure, within the `testsuite` directory. Since a test suite has no public header files, there is no `include` directory in a test suite.

Rule 6

Public header files always go into the `include` directory or a subdirectory thereof. To avoid name clashes and for better comprehensibility, the `include` directory may contain subdirectories. For libraries, the include directory usually contains a subdirectory with the same name as the library.

Rule 7

All implementation files and non-public header files go into the `src` directory.

Rule 8

All project files, make files and other support files not containing source code go directly into the project directory.

Recommendation 4

If there is extra documentation for a project (e.g. specifications or standard documents), this documentation goes into a directory named `doc`.

4.4 Whitespace

Rule 9

In a header or implementation file, introductory comment, include guards, `#include` block, `using` block and function definitions are separated by

two empty lines. This makes it easier to visually distinguish the various elements.

Rule 10

The last line in an implementation or header file must be terminated by a newline. This is required by the C++ standard, but not enforced by most compilers.

4.5 Comments

Rule 11

Every file that contains source code must be documented with an introductory comment that provides information on the file name, its version and its contents.

Rule 12

All files must include copyright information.

Rule 13

All comments are written in English.

Rule 14

Write a comment for every class, public/protected function and public/protected `enum`/`typedef`/`struct`. The standardization of comments makes it possible to automatically generate reference documentation from source code. This is used to keep source code and documentation together up-to-date. The format of these comments is described in the Appendix.

Rule 15

Use `//` for comments.

It is necessary to document source code. This should be compact and easy to find. By properly choosing names for variables, functions and classes and by properly structuring the code, there is less need for comments within the code.

Note that comments in header files are meant for the users of classes, while comments in implementation files are meant for those who maintain the classes.

All our code must be copyright marked. If the code has been developed over a period of years, each year must be stated.

Comments are often said to be either strategic or tactical. A strategic comment describes what a function or section of code is intended to do, and is placed before this code. A tactical comment describes what a single line of code is intended to do, and is placed, if possible, at the end of this line. Unfortunately, too many tactical comments can make code unreadable. For this reason, it is recommended to primarily use strategic comments, unless trying to explain very complicated code.

If the characters `//` are consistently used for writing comments, then the combination `/* */` could be used to make comments out of entire sections of code during the development and debugging phases. C++, however, does not allow comments to be nested using `/* */`. So it is better to comment out large sections of code using the preprocessor (`#if 0 ... #endif`).

Source files containing commented-out code sections must not be checked in to the SCM repository, as this would confuse other developers working with that code.

Example 1

Boilerplate text to be included at the beginning of every header or implementation file:

```
//
// <FileName>.h
//
// Library: <LibraryName>
// Package: <PackageName>
// Module:  <ModuleName>
//
// <One or two sentences describing what the header file is for. Can
// be omitted in source (.cpp) files.>
//
// Copyright (c) 2023, Applied Informatics Software Engineering GmbH.
// All rights reserved.
//
// <License Notice>
//
```

4.6 Header Files

Rule 16

Every header file must contain a mechanism that prevents multiple inclusions of the file.

Rule 17

When the following kinds of definitions are used (in implementation files or in other header files), they must be included as separate header files:

- classes that are used as base classes,
- classes that are used as member variables,
- classes that appear as return types or as argument types in function/member function prototypes.
- function prototypes for functions/member functions used in inline member functions that are defined in the file.

Rule 18

Definitions of classes that are only accessed via pointers (*) or references (&) shall not be included as header files. Forward declarations shall be used instead. The only exceptions are classes that are in another namespace.

Rule 19

Never specify relative paths (containing "." and "..") in `#include` directives.

Rule 20

Every implementation file must include the relevant files that contain:

- declarations of types and functions used in the functions that are implemented in the file.
- declarations of variables and member functions used in the functions that are implemented in the file.

Rule 21

Every implementation file must, before any other header files, include its corresponding header file. This ensures that header files are self-sufficient.

Rule 22

Use the directive `#include "filename.h"` for user-prepared include files.

Rule 23

Use the directive `#include <filename.h>` for system and compiler-supplied header files only.

Rule 24

Compiler include paths always point to the `include` directory in a project directory. Therefore, if a header file is located in a subdirectory of the include directory, it must be included with `#include`

`"dir/filename.h"`. As a positive side effect, this avoids name clashes if different projects have a header file with the same name.

The easiest way to avoid multiple includes of files is by using an `#ifndef/#define` block in the beginning of the file and an `#endif` at the end of the file.

The number of files included should be minimized. If a file is included in an include file, then every implementation file that includes the second include file must be re-compiled whenever the first file is modified. A simple modification in one include file can make it necessary to re-compile a large number of files.

When only referring to pointers or references to types defined in a file, it is often not necessary to include that file. It may suffice to use a forward declaration to inform the compiler that the class exists. Another alternative is to precede each declaration of a pointer to the class with the keyword `class`.

Example 2

Using include guards to prevent multiple inclusions of a header file:

```
#ifndef Foundation_SharedPtr_INCLUDED
#define Foundation_SharedPtr_INCLUDED


#include "Poco/Foundation.h"


...


#endif // Foundation_SharedPtr_INCLUDED
```

> *Most C++ compilers support the* `#pragma once` *preprocessor directive to prevent multiple inclusions of the same header file. However, this is not part of the C++ standard and should therefore be not used.*

5 Names

Rule 25

Every class library must define a unique namespace for its globally visible identifiers.

Rule 26

The identifier of every globally visible class, enumeration type, type definition, function, constant, and variable in a class library must be inside the class library's namespace.

Rule 27

The names of variables, and functions begin with a lowercase letter.

Rule 28

The names of abstract data types, classes, structures, types, and enumerated types begin with an uppercase letter.

Rule 29

In names that consist of more than one word, the words are written together and each word that follows the first begins with an uppercase letter. [1]

Rule 30

Global names must not begin with one or two underscores ("_" or "__"). This is forbidden by the C++ standard as such names are reserved for use by the compiler.

Rule 31

With the exceptions of constants and enumeration values, there are no underscores in names (after the first character of the name). [2]

Rule 32

Do not use type names that differ only by the use of uppercase and lowercase letters.

[1] This is also known as *camel-case*.
[2] The leading underscore in names of private and protected member variables (see Rule 33) does not count here.

Rule 33

The name of a private or protected member variable begins with a single underscore.[3] This is the only underscore in such a name (see Rule 31).

Rule 34

The names of constants and enumeration values are all uppercase. In names that consist of more than one word, these words are separated by underscore characters.

Rule 35

Encapsulate global variables and constants, enumerated types, and typedefs in a class.

Recommendation 5

Names should be self-descriptive yet as brief as possible. Cryptic abbreviated names are as hard to read as well as too long ones.

Recommendation 6

Names should be pronounceable. It is hard to discuss something that cannot be pronounced.

Recommendation 7

Names should not include abbreviations that are not generally accepted. A list of generally accepted abbreviations can be found in the Appendix.

Recommendation 8

A variable holding a pointer should be prefixed with “p”. A variable holding an iterator should be prefixed with “it”.

Recommendation 9

Apart from the cases in Recommendation 8, avoid encodings in names. Encoded names require deciphering. Especially, avoid type-encoded variable names (Petzold-style Hungarian notation like `lpcszName`).

[3] There are different opinions regarding the validity of leading underscores in class member names. The C++ standard explicitly states that leading underscores are not allowed in global names. Class members are not global names, therefore this rule does not apply to them. Experience with a variety of different compilers does not show any problems. A commonly used alternative is to append an underscore at the end of a member variable name.

Recommendation 10

Be consistent. If something is a `name` it should be a `name` everywhere it is used (not, e.g. an `id` somewhere else).

Example 3

Identifier names in a class declaration (comments have been removed):

```
class Net_API HTTPRequest: public HTTPMessage
{
public:
    HTTPRequest();
    HTTPRequest(const std::string& version);
    HTTPRequest(const std::string& method, const std::string& uri);
    virtual ~HTTPRequest();
    void setMethod(const std::string& method);
    const std::string& getMethod() const;
    void write(std::ostream& ostr) const;
    void read(std::istream& istr);

    static const std::string HTTP_GET;
    static const std::string HTTP_HEAD;
    static const std::string HTTP_PUT;
    static const std::string HTTP_POST;
    static const std::string HTTP_OPTIONS;
    static const std::string HTTP_DELETE;
    static const std::string HTTP_TRACE;
    static const std::string HTTP_CONNECT;

private:
    enum Limits
    {
        MAX_METHOD_LENGTH  = 32,
        MAX_URI_LENGTH     = 4096,
        MAX_VERSION_LENGTH = 8
    };

    std::string _method;
    std::string _uri;
};
```

6 Style

6.1 Classes

Rule 36

The `public`, `protected`, and `private` sections of a class are declared in that order (the `public` section is declared before the `protected` section which is declared before the `private` section).

Rule 37

With the exception of trivial member functions in class templates, member functions are never defined within the class definition.

Rule 38

Inline functions and class template member functions are defined in a separate block following the class declaration.

Recommendation 11

If a member variable has an accessor, but not a mutator, the accessor has the same name as the variable, sans the underscore prefix. If a member variable has both an accessor and a mutator, the accessor name has a `get` prefix and the mutator name has a `set` prefix.

By placing the public section first, everything that is of interest to a user is gathered in the beginning of the class definition. The protected section may be of interest to designers when considering inheriting from the class. The private section contains details that should have the least general interest.

A member function that is defined within a class definition automatically becomes inline. Class definitions are less compact and more difficult to read when they include definitions of member functions. It is easier for an inline member function to become an ordinary member function if the definition of the inline function is placed outside of the class definition.

Example 4

Defining an inline function:

```
class Foundation_API UUID
{
public:
    ...
    bool operator == (const UUID& uuid) const;
    bool operator != (const UUID& uuid) const;
    bool operator <  (const UUID& uuid) const;
```

```
    bool operator <= (const UUID& uuid) const;
    bool operator >  (const UUID& uuid) const;
    bool operator >= (const UUID& uuid) const;
    ...
};


//
// inlines
//
inline bool UUID::operator == (const UUID& uuid) const
{
    return compare(uuid) == 0;
}


inline bool UUID::operator != (const UUID& uuid) const
{
    return compare(uuid) != 0;
}


...
```

Example 5

Accessors and mutators:

```
class Property
{
public:
    Property(const std::string& name);
    Property(const Property& prop);
    ~Property();
    Property& operator = (const Property&);
    const std::string& name();
    void setValue(const std::string& value);
    const std::string& getValue() const;

private:
    Property();

    std::string _name;
    std::string _value;
};
```

6.2 Functions

Recommendation 12

When declaring functions, the leading parenthesis and the first argument (if any) are written on the same line as the function name. If space and readability permit, other arguments and the closing parenthesis may also be written on the same line as the function name. Otherwise, each additional argument is written on a separate line, indented with a single tab (with the closing parenthesis directly after the last argument).

Example 6

Long function declarations:

```
DateTime& assign(
    int year,
    int month,
    int day,
    int hour = 0,
    int minute = 0,
    int second = 0,
    int millisecond = 0,
    int microseconds = 0);
```

Recommendation 13

Always write the left parenthesis directly after a function name. There is no space between the function name, the opening brace and the first argument declaration. Also there is no space between the last argument and the closing parenthesis.

6.3 Templates

Rule 39

The `template` keyword, together with the template argument list, is written on a separate line before the following class or function definition.

6.4 Compound Statements

Recommendation 14

Braces ("`{}`") enclosing a block are placed in the same column, on separate lines directly before and after the block.

The placement of braces seems to have been the subject of the greatest debate concerning the appearance of both C and C++ code. We recommend a style that, in our opinion, gives the most readable code. Other styles may well provide more compact code.

6.5 Flow Control Statements

Recommendation 15

There is always a space between the flow control statement's keyword and the opening parenthesis of the control expression. There is no space between the opening parenthesis and the expression. There is also no space between the expression and the closing parenthesis.

Recommendation 16

The flow control primitives `if`, `else`, `while`, `for` and `do` should be followed by a block, even if it is an empty block, or consisting of only one statement.

At times, everything that is done in a loop may be easily written on one line in the loop statement itself. It may then be tempting to conclude the statement with a semicolon at the end of the line. This may lead to misunderstanding since, when reading the code, it is easy to miss such a semicolon. Also, the semicolon could be deleted by accident, leading to a hard-to-find bug. It seems to be better, in such cases, to place an empty block after the statement to make completely clear what the code is doing. Even better is to avoid this style altogether.

In certain cases, a code that is more compact might be better readable than code with many blocks containing only a single statement. As a general rule, readability must always be preferred to strict style adherence.

Example 7

Blocks and single statements:

```
int ASCIIEncoding::convert(int ch, unsigned char* bytes, int length) const
{
    if (ch >= 0 && ch <= 127)
    {
        *bytes = (unsigned char) ch;
        return 1;
    }
    else return 0;
}
```

Example 8

Single-line flow control statements can improve readability by keeping the code more compact:

```
void MessageHeader::splitParameters(
    const std::string& s,
    std::string& value,
    NameValueCollection& parameters)
{
    value.clear();
    parameters.clear();
    std::string::const_iterator it  = s.begin();
    std::string::const_iterator end = s.end();
    while (it != end && isspace(*it)) ++it;
    while (it != end && *it != ';') value += *it++;
    Poco::trimRightInPlace(value);
    if (it != end) ++it;
    splitParameters(it, end, parameters);
}
```

6.6 Pointers and References

Recommendation 17

The dereference operator "`*`" and the address-of operator "`&`" should be directly connected with the type names in declarations and definitions.

The characters "`*`" and "`&`" should be written together with the types of variables instead of with the names of variables in order to emphasize that they are part of the type definition. Instead of saying that `*i` is an `int`, say that `i` is an `int*`.

Traditionally, C recommendations indicate that "`*`" should be written together with the variable name, since this reduces the probability of making a mistake when declaring several variables in the same declaration statement (the operator "`*`" only applies to the variable on which it operates). Since the declaration of several variables in the same statement is not recommended, however, such an advice is unneeded.

Rule 40

Never declare more than one variable in a single statement.

6.7 Miscellaneous

Recommendation 18

Do not use spaces around "`.`" or "`->`", nor between unary operators and operands.

Recommendation 19

Indentation is done with tabs and the number of spaces in a tab is set to four (4) in the editor.

Code that is indented with spaces ends up looking messy because after the third indentation step, developers often lose track of the correct number of spaces.

7 Classes

7.1 Access Rights

Rule 41

Do not specify `public` or `protected` member data in a class.

The use of `public` variables is discouraged for the following reasons:

1. A public variable represents a violation of one of the basic principles of object-oriented programming, namely, encapsulation of data. For example, if there is a class of the type `BankAccount`, in which `accountBalance` is a public variable, the value of this variable may be changed by any user of the class. However, if the variable has been declared private, its value may be changed only by the member functions of the class.
2. An arbitrary function in a program can change public data, leading to errors that are difficult to locate.
3. If public data is avoided, its internal representation may be changed without users of the class having to modify their code. A principle of class design is to maintain the stability of the public interface of the class. The implementation of a class should not be a concern for its users.

The use of protected variables in a class is not recommended, since they become visible to its derived classes. The names of types or variables in a base class may then not be changed since the derived classes may depend on them. If a derived class, for some reason, must access data in its base class, one solution may be to make a special protected interface in the base class, containing functions that return private data. This solution is not likely to imply any degradation of performance if the functions are defined inline.

The use of structs is recommended for trivial data-only structures that do not have class invariants (e.g., parameter packs).

7.2 Inline Functions

Recommendation 20

Access functions (accessors) that simply return the value of a member variable are inline.

Recommendation 21

Forwarding functions are inline.

Recommendation 22

Constructors and destructors should not be inline.

The normal reason for declaring a function inline is to improve its performance.

Small functions, such as access functions, which only return the value of a member of the class and so-called forwarding functions that invoke another function should normally be inline.

Correct usage of inline functions may also lead to reduced code size.

Warning: functions invoking other inline functions often become too complex for the compiler to be able to make them inline despite their apparent smallness.

This problem is especially common with constructors and destructors. A constructor always invokes the constructors of its base classes and member data before executing its own code. Inline constructors and destructors are best avoided!

7.3 Friends

Recommendation 23

Friends of a class should be used to provide additional functions that are best kept outside of the class.

Operations on an object are sometimes provided by a collection of classes and functions.

A friend is a nonmember of a class that has access to the nonpublic members of the class. Friends offer an orderly way of getting around data encapsulation for a class. A friend class can be advantageously used to provide functions that require data that is not normally needed by the class.

Suppose there is a list class that needs a pointer to an internal list element in order to iterate through the class. This pointer is not needed for other operations on the list. There may then be reason, such as obtaining smaller list objects, for a list object not to store a pointer to the current list element and instead to create an iterator, containing such a pointer, when it is needed.

One problem with this solution is that the iterator class normally does not have access to the data structures that are used to represent the list (since we also recommend private member data).

By declaring the iterator class as a friend, this problem is avoided without violating data encapsulation.

Friends are good if used properly. However, the use of many friends can indicate that the modularity of the system is poor.

7.4 Const Member Functions

Rule 42

A member function that does not affect the state of an object (its instance variables) is declared `const`.

Rule 43

If the behavior of an object is dependent on data outside the object, this data is not modified by `const` member functions.

Member functions declared as `const` must not modify member data and are the only functions that may be invoked on a `const` object. (Such an object is clearly unusable without `const` methods.) A `const` declaration is an excellent insurance that objects will not be modified (mutated) when they should not be. A great advantage of C++ is the ability to overload functions with respect to their const-ness. Two member functions may have the same name where one is const and the other one is not.

Non-const member functions are sometimes invoked as so-called “lvalues” (as a location value at which a value may be stored). A const member function may never be invoked as “lvalue”.

The behavior of an object can be affected by data outside the object. Such data must not be modified by a `const` member function.

In rare cases, it is desirable to modify member variables in a `const` object. Such member variables must be declared as `mutable`.

7.5 Constructors and Destructors

Rule 44

Every class that has pointers as instance variables must declare a copy constructor (and an assignment operator). The copy constructor may be private or deleted (`= delete`), in which case no implementation is provided.

A class that defines a copy constructor should also define a move constructor.

Rule 45

All classes that can be used as base classes must define a virtual destructor.

Rule 46

If a class has at least one virtual function, it must also have a virtual destructor.

Rule 47

Every class must have a default (no-argument) constructor and a destructor. Unless the default constructor is private, an implementation (even if empty) must be provided for the default constructor. An empty implementation should be provided by defining the constructor as `= default` rather than explicitly writing an empty implementation.

Rule 48

Destructors must not throw.

In C++11, destructors are implicitly specified as *noexcept(true)*. So a destructor that throws will cause the program to terminate immediately.

Recommendation 24

Wrap the code in destructors in a *try … catch (…)* block if the code may do anything that could throw. In order to not silently swallow exceptions that may be thrown (and which may be a hint for errors), POCO provides the *poco_unexpected()* macro, which should be used in the exception handler. In debug builds, this will write a message to standard output and, if running under a debugger, trigger a breakpoint.

Example 9

```
MyClass::~MyClass
{
    try
    {
        cleanup();
    }
    catch (...)
    {
        poco_unexpected();
    }
}
```

Recommendation 25

Define and initialize member variables in the same order. Prefer initialization to assignment in constructors. Prefer initialization in the class definition to initialization in a constructor, if possible.

Recommendation 26

Do not call a virtual function from a constructor or destructor.

Recommendation 27

Avoid the use of global objects in constructors and destructors.

Recommendation 28

Use `explicit` for single argument non-copy constructors. This helps to avoid ambiguities caused by implicit type conversions.

A copy constructor is recommended to avoid surprises when an object is initialized using an object of the same type. If an object manages the allocation and deallocation of an object on the heap (the managing object has a pointer to the object to be created by the class' constructor), only the value of the pointer will be copied. This can lead to two invocations of the destructor for the same object (on the heap), probably resulting in a run-time error.

The corresponding problem exists for the assignment operator ("=").

If a class, having virtual functions but without virtual destructors, is used as a base class, there may be a nasty surprise if pointers to the class are used. If such a pointer is assigned to an instance of a derived class and if delete is then used on this pointer, only the base class' destructor will be invoked. If the program depends on the derived class' destructor being invoked, the program will fail.

In connection with the initialization of statically allocated objects, it is not guaranteed that other static objects will be initialized (for example, global objects). This is because the order of initialization of static objects that are defined in different compilation units is not defined in the language definition.

Example 10

Initialization of member variables in a constructor:

```
class Property
{
public:
    Property(const std::string& name, const std::string& value);
    ...
private:
    std::string _name;
    std::string _value;
};


// Good:
Property::Property(const std::string& name, const std::string& value):
    _name(name),
    _value(value)
{
}

// Bad:
Property::Property(const std::string& name, const std::string& value)
{
    _name = name;
    _value = value;
}
```

7.6 Assignment Operator

Rule 49

Every class that has pointers as instance variables must declare an assignment operator along with a copy constructor. The assignment operator may be private or deleted (`= delete`), in which case no implementation is provided.

Rule 50

An assignment operator performing a destructive action must be protected from performing this action on the object upon which it is operating.

Recommendation 29

An assignment operator shall return a non-const reference to the assigning object.

Recommendation 30

Whenever possible, the implementation of an assignment operator shall use the `swap()` operation to provide for strong exception safety.

An assignment is not inherited like other operators. If an assignment operator is not explicitly defined, then one is automatically defined instead. Such an assignment operator does not perform bit-wise copying of member data; instead, the assignment operator (if defined) for each specific type of member data is invoked. Bit-wise copying is only performed for member data having primitive types.

One consequence of this is that bit-wise copying is performed for member data having pointer types. If an object manages the allocation of the instance of an object pointed to by a pointer member, this will probably lead to problems: either by invoking the destructor for the managed object more than once or by attempting to use the deallocated object.

If an assignment operator is overloaded, the programmer must make certain that the base class' and the members' assignment operators are run.

A common error is assigning an object to itself. Normally, destructors for instances allocated on the heap are invoked before assignment takes place. If an object is assigned to itself, the values of the instance variables will be lost before they are assigned. This may well lead to strange run-time errors.

Example 11

Implementing the assignment operator using `swap()`:

```
Foo& Foo::operator = (const Foo& foo)
{
```

```
    Foo tmp(foo);
    swap(tmp);
    return *this;
}
```

7.7 Operator Overloading

Recommendation 31

Use operator overloading sparingly and in a uniform manner.

Recommendation 32

When two operators are opposites (such as == and `!`=), it is appropriate to define both.

Recommendation 33

Do not overload `operator &&`, `operator ||` or `operator ,` (comma).

Recommendation 34

Do not overload type conversion (cast) operators.

Operator overloading has both advantages and disadvantages. One advantage is that code using a class with overloaded operators can be written more compactly (more readably). Another advantage is that the semantics can be both simple and natural. One disadvantage in overloading operators is that it is easy to misunderstand the meaning of an overloaded operator (if the programmer has not used natural semantics). The extreme case, where the plus-operator is re-defined to mean minus and the minus-operator is re-defined to mean plus, probably will not occur very often, but more subtle cases are conceivable.

Designing a class library is like designing a language! If you use operator overloading, use it in a uniform manner; do not use it if it can easily give rise to misunderstanding.

7.8 Member Function Return Types

Rule 51

A public member function must never return a non-const reference or pointer to member data.

A public member function must never return a non-const reference or pointer to data outside an object, unless the object shares the data with other objects.

By allowing a user direct access to the private member data of an object, this data may be changed in ways not intended by the class designer. This may lead to reduced confidence in the designer's code: a situation to be avoided.

7.9 Inheritance

Recommendation 35

Avoid inheritance for parts-of relations. Prefer composition to inheritance.

Recommendation 36

Give derived classes access to class type member data by declaring protected access functions.

A common mistake is to use multiple inheritance for parts-of relations (when an object consists of several other objects, these are inherited instead of using instance variables. This can result in strange class hierarchies and less flexible code. In C++ there may be an arbitrary number of instances of a given type; if inheritance is used, direct inheritance from a class may only be used once.

A derived class often requires access to base class member data in order to create useful member functions. The advantage of using protected member functions is that the names of base class member data are not visible in the derived classes and thus may be changed. Such access functions should only return the values of member data (read-only access).

The guiding assumption is that those who use inheritance know enough about the base class to be able to use the private member data correctly, while not referring to this data by name. This reduces the coupling between base classes and derived classes.

8 Class Templates

Recommendation 37

Do not attempt to create an instance of a class template using a type that does not define the member functions that the class template, according to its documentation, requires.

Recommendation 38

Take care to avoid multiple definitions of overloaded functions in conjunction with the instantiation of a class template.

Before C++20 (concepts), it is not possible in C++ to specify requirements for type arguments for class templates and function templates. This may imply that the type chosen by the user does not comply with the interface as required by the template. For example, a class template may require that a type argument have a comparison operator defined.

Another problem with type templates can arise for overloaded functions. If a function is overloaded, there may be a conflict if the element type appears explicitly in one of these. After instantiation, there may be two functions which, for example, have the type int as an argument. The compiler may complain about this, but there is a risk that the designer of the class does not notice it. In cases where there is a risk for multiple definition of member functions, this must be carefully documented.

Recommendation 39

Only trivial member functions of a class template should be defined inside the class definition. Member functions with more than 2-3 lines of code should be defined outside the class template.

9 Functions

Unless otherwise stated, the following rules also apply to member functions.

9.1 Function Arguments

Rule 52

Do not use unspecified function arguments (ellipsis notation).

Recommendation 40

Avoid const pass-by-value function parameters in function declarations. They are pointless.

Recommendation 41

Avoid functions with too many (rule of thumb: more than five) arguments. Functions having long argument lists look complicated, are difficult to read, and can indicate poor design. In addition, they are difficult to use and to maintain.

Recommendation 42

Prefer references to pointers. Use a pointer only if the argument could be null or if using a reference would always require dereferencing a pointer when calling the function.

Recommendation 43

Use constant references (`const &`) instead of call-by-value, unless using a pre-defined data type or a pointer.

By using references instead of pointers as function arguments, code can be made more readable, especially within the function. A disadvantage is that it is not easy to see which functions change the values of their arguments. Member functions storing pointers provided as arguments should document this clearly by declaring the argument as a pointer instead of as a reference. This simplifies the code, since it is normal to store a pointer member as a reference to an object.

There are no null references in C++. This means that an object must have been allocated before passing it to a function. The advantage of this is that it is not necessary to test the existence of the object within the function.

C++ invokes functions using call-by-value semantics. This means that the function arguments are copied to the stack via invocations of copy constructors, which, for large objects, reduces performance. In addition,

destructors will be invoked when exiting the function. `const &` arguments mean that only a reference to the object in question is placed on the stack (call-by-reference) and that the object's state (its instance variables) cannot be modified. (At least some const member functions are necessary for such objects to be at all useful.)

9.2 Function Overloading

Recommendation 44

When overloading functions, all variations should have the same semantics (be used for the same purpose).

Overloading of functions can be a powerful tool for creating a family of related functions that only differ as to the type of data provided as arguments. If not used properly (such as using functions with the same name for different purposes), they can, however, cause considerable confusion.

Template functions can be a good alternative to overloading.

9.3 Formal Arguments

Rule 53

The names of formal arguments to functions must be specified and must be the same both in the function declaration and in the function definition.

The names of formal arguments may be specified in both the function declaration and the function definition in C++, even if the compiler ignores those in the declaration. Providing names for function arguments is a part of the function documentation. The name of an argument clarifies how the argument is used and reduces the need to include comments in a function definition. It is also easier to refer to an argument in the documentation of a class if it has a name.

Exception: the name of an unused function argument can be omitted, both in the declaration and in the definition.

9.4 Return Types and Values

Rule 54

Always specify the return type of a function explicitly. Modern C++ compilers expect this anyway.

Rule 55

A public function must never return a reference or a pointer to a local variable.

If a function returns a reference or a pointer to a local variable, the memory to which it refers will already have been deallocated when this reference or pointer is used. The compiler may or may not give a warning for this.

9.5 Inline Functions

Rule 56

Do not use the preprocessor directive `#define` to obtain more efficient code; instead, use `inline` functions.

Rule 57

Use inline functions when they are really needed.

Inline functions have the advantage of often being faster to execute than ordinary functions. The disadvantage in their use is that the implementation becomes more exposed, since the definition of an inline function must be placed in an include file for the class, while the definition of an ordinary function may be placed in its own separate file.

A result of this is that a change in the implementation of an inline function can require comprehensive re-compiling when the include file is changed.

The compiler is not compelled to actually make a function inline. The decision criteria for this differ from one compiler to another.

9.6 Temporary Objects

Recommendation 45

Minimize the number of temporary objects that are created as return values from functions or as arguments to functions.

Temporary objects are often created when objects are returned from functions or when objects are given as arguments to functions. In either case, a constructor for the object is first invoked; later, a destructor is invoked. Large temporary objects make for inefficient code. In some cases, errors are introduced when temporary objects are created. It is important to keep this in mind when writing code. It is especially inappropriate to have pointers to temporary objects, since the lifetime of a temporary object is undefined.

9.7 General

Recommendation 46

Avoid long and complex functions.

Long functions have disadvantages:

1. If a function is too long, it can be difficult to comprehend. Generally, it can be said that a function should not be longer than 60 lines (or one page), since that is about how much can be comprehended at one time. This is also what can be displayed in an editor window without scrolling.
2. If an error situation is discovered at the end of an extremely long function, it may be difficult for the function to clean up after itself and to "undo" as much as possible before reporting the error to the calling function. By always using short functions, such an error can be more exactly localized.

Complex functions are difficult to test. If a function consists of 15 nested if statements, then there are 2**15 (or 32768) different branches to test in a single function.

Nesting of control structures should not exceed three levels.

10 Constants

Rule 58

Constants are defined using `const` or `enum`; never using `#define`.

The preprocessor performs a textual substitution for macros in the source code that is then compiled. This has a number of negative consequences. For example, if a constant has been defined using `#define`, the name of the constant is not recognized in many debuggers. If the constant is represented by an expression, this expression may be evaluated differently for different instantiations, depending on the scope of the name. In addition, macros are, at times, incorrectly written.

Rule 59

Avoid the use of numeric values in code; use symbolic values instead.

Numerical values in code (*magic numbers*) should be viewed with suspicion. They can be the cause of difficult problems if and when it becomes necessary to change a value. A large amount of code can be dependent on such a value never changing, the value can be used at a number of places in the code (it may be difficult to locate all of them), and values as such are rather anonymous (it may be that every “2” in the code should not be changed to a “3”).

From a portability point of view, absolute values may be the cause of more subtle problems. The type of a numeric value is dependent on the implementation. Normally, the type of a numeric value is defined as the smallest type able to hold the value.

11 Variables

Rule 60

Variables are declared with the smallest possible scope.

Rule 61

Each variable is declared in a separate declaration statement.

Rule 62

Every variable that is declared is given an initial value before it is used.

Rule 63

If possible, always use initialization instead of assignment.

A variable ought to be declared with the smallest possible scope to improve the readability of the code and so that variables are not unnecessarily allocated. When a variable that is declared at the beginning of a function is used somewhere in the code, it is not easy to directly see the type of the variable. In addition, there is a risk that such a variable is inadvertently hidden if a local variable, having the same name, is declared in an internal block.

Many local variables are only used in special cases which seldom occur. If a variable is declared at the outer level, memory will be allocated even if it is not used. In addition, when variables are initialized upon declaration, more efficient code is obtained than if values are assigned when the variable is used.

A variable must always be initialized before use. Normally, the compiler gives a warning if a variable is undefined. It is then sufficient to take care of such cases. Instances of a class are usually initialized even if no arguments are provided in the declaration (the empty constructor is invoked). To declare a variable that has been initialized in another file, the keyword extern is always used.

By always initializing variables, instead of assigning values to them before they are first used, the code is made more efficient since no temporary objects are created for the initialization. For objects having large amounts of data, this can result in significantly faster code.

12 Pointers and References

Rule 64

Do not compare a pointer to `NULL` or assign `NULL` to a pointer; use `nullptr` instead. Also prefer `nullptr` to `0` (which was used before C++11).

Recommendation 47

Pointers to pointers should whenever possible be avoided.

Recommendation 48

Use a `typedef` to simplify program syntax when declaring function pointers.

According to the ANSI-C standard, `NULL` is defined either as `(void*)0` or as 0. If this definition remains in ANSI-C++, problems may arise. If `NULL` is defined to have the type `void*`, it cannot be assigned an arbitrary pointer without an explicit type conversion.

Pointers to pointers normally ought not be used. Instead, a class should be declared, which has a member variable of the pointer type. This improves the readability of the code and encourages data abstraction. By improving the readability of code, the probability of failure is reduced. One exception to this rule is represented by functions providing interfaces to other languages (such as C). These are likely to only allow pre-defined data types to be used as arguments in the interface, in which case pointers to pointers are needed. Another example is the second argument to the main function, which must have the type `char*[]`.

A function that changes the value of a pointer that is provided as an argument should declare the argument as having the type reference to pointer (e.g. `char*&`).

13 Type Conversions

Rule 65

Use C++ style casts (`dynamic_cast<>`, `static_cast<>`, `reinterpret_cast<>`, `const_cast<>`) instead of old-style C casts for all pointer conversions. This makes type conversions more explicit.

Rule 66

Avoid the use of `reinterpret_cast<>`.

Exception: low-level code interfacing the operating system or hardware, where such a cast might be necessary.

14 Flow Control

Rule 67

The code following a `case` label must always be terminated by either a `break` statement, a `return` statement or a `throw` statement.

If the code following a case label is not terminated by break, the execution continues after the next case label. This means that poorly tested code can be erroneous yet still seem to work.

Rule 68

A `switch` statement must always contain a `default` branch that handles unexpected cases. The `default` label is always the last label in a `switch` statement.

Rule 69

Case labels (and the `default` label) in a `switch` statement always have the same indentation level as the `switch` statement.

Rule 70

Never use `goto`.

`goto` breaks the control flow and can lead to code that is difficult to comprehend. In addition, there are limitations for when `goto` can be used. For example, it is not permitted to jump past a statement that initializes a local object having a destructor.

Recommendation 49

The choice of loop construct (`for`, `while` or `do while`) should depend on the specific use of the loop.

Each loop construct has a specific usage. A `for` loop is used only when the loop variable is increased by a constant amount for each iteration and when the termination of the loop is determined by a constant expression. In other cases, `while` or `do while` should be used. When the terminating condition can be evaluated at the beginning of the loop, `while` should be used; `do while` is used when the terminating condition is best evaluated at the end of the loop.

Recommendation 50

Prefer C++ standard library algorithms over explicit loops when iterating over the members of a container.

Recommendation 51

Use `break` to exit a loop if this avoids the use of flags.

Recommendation 52

Use inclusive lower and exclusive upper limits.

Instead of saying that `x` is in the interval `x >= 23` and `x <= 42`, use the limits `x >= 23` and `x < 43`. The following important claims then apply:

- The size of the interval between the limits is the difference between the limits.
- The limits are equal if the interval is empty.
- The upper limit is never less than the lower limit.

Being consistent in this regard helps to avoid many difficult bugs.

Example 12

Canonical `for` loop:

```
for (int i = 0; i < size; ++i)
{
    ...
}
```

Example 13

Canonical iterator `for` loop:

```
for (Container::const_iterator it = cont.begin(); it != cont.end(); ++it)
{
    ...
}
```

Example 14

`switch` statement formatting:

```
switch (expression)
{
case foo:
    ...
    break;
case bar:
    …
    break;
default:
    poco_bugcheck();
}
```

15 Expressions

Recommendation 53

Use parentheses to clarify the order of evaluation for operators in expressions.

There are a number of common pitfalls having to do with the order of evaluation for operators in an expression. Binary operators in C++ have associativity (either leftwards or rightwards) and precedence. If an operator has leftwards associativity and occurs on both sides of a variable in an expression, then the variable belongs to the same part of the expression as the operator on its left side.

Recommendation 54

Prefer prefix increment/decrement to the postfix variants.

In doubtful cases, parentheses are always to be used to clarify the order of evaluation.

Another common mistake is to confuse the assignment operator and the equality operator. Since the assignment operator returns a value, it is entirely permitted to have an assignment statement instead of a comparison expression. This, however, most often leads straight to an error.

C++ allows the overloading of operators, something that can easily become confusing. For example, the operators << (shift left) and >> (shift right) are often used for input and output. Since these were originally bit operations, it is necessary that they have higher priority than relational operators. This means that parentheses must be used when outputting the values of logical expressions.

Iterators overload the increment and decrement operators. Using the postfix increment/decrement operators on an iterator causes the construction of a temporary object. Therefore prefer the prefix operators.

Recommendation 55

Always have non-lvalues on the left side (`0 == i` instead of `i == 0`).

That way compiler works for you. It takes a while to get used to it, but it's worth it. For intuitiveness sake, this should be done only for == and `!=` comparison.

16 Memory and Resources

Rule 71

Do not redefine the global `new` and `delete` operators.

Rule 72

When overloading the `new` operator for a class, always overload the `delete` operator too.

Rule 73

Do not use `malloc()`, `realloc()` or `free()`.

Rule 74

Always use the array delete operator (`delete []`) when deallocating arrays.

Rule 75

Use smart pointers (`std::unique_ptr`, `Poco::AutoPtr`) or shared pointers (`std::shared_ptr`, `Poco::SharedPtr`) instead of raw pointers wherever possible.

Recommendation 56

Use the RAII (Resource Acquisition Is Initialization) idiom wherever possible.

Recommendation 57

Avoid global data if at all possible.

Recommendation 58

Do not allocate memory and expect that someone else will deallocate it later.

Recommendation 59

Always assign a new value to a pointer pointing to deallocated memory.

In C++ data can be allocated statically, dynamically on the stack, or dynamically on the heap. There are three categories of static data: global data, global class data, and static data local to a function.

In C `malloc()`, `realloc()` and `free()` are used to allocate memory dynamically on the heap. This may lead to conflicts with the use of the `new` and `delete` operators in C++.

It is forbidden to:

1. invoke `delete` for a pointer obtained via `malloc()`/`realloc()`,
2. invoke `malloc()`/`realloc()` for objects having constructors,
3. invoke `free()` for anything allocated using `new`.

Avoid whenever possible the use of `malloc()`, `realloc()` and `free()`.

If an array a having a type `T` is allocated, it is important to invoke `delete` in the correct way. Only writing `delete a;` will result in the destructor being invoked only for the first object of type `T`. By writing `delete [] a;` the destructor will be invoked for all objects that have been allocated earlier.

Static data can cause several problems. In an environment where parallel threads execute simultaneously, they can make the behavior of code unpredictable, since functions having static data are not reentrant.

One difference between ANSI-C and C++ is in how constants are declared. If a variable is declared as a constant in ANSI-C, it has the storage class extern (global). In C++, however, it normally has the storage class static (local). The latter means that a new instance of the constant object is created each time a file includes the file which contains the declaration of the object, unless the variable is explicitly declared extern in the include file.

An extern declaration in C++ does not mean that the variable is initialized; there must be a definition for this in a definition file. Static constants that are defined within a class are always external and must always be defined separately.

It may, at times, be tempting to allocate memory for an object using `new`, expecting someone else to deallocate the memory. For instance, a function can allocate memory for an object that is then returned to the user as the return value for the function. There is no guarantee that the user will remember to deallocate the memory and the interface with the function then becomes considerably more complex.

Pointers that point to deallocated memory should either be set to 0 or be given a new value to prevent access to the released memory. This can be a very difficult problem to solve when there are several pointers pointing to the same memory, since C++ has no garbage collection.

17 Types and Namespaces

Rule 76

Do not place `using namespace` in a header file or before an `#include`.

Rule 77

A source file must not define more than one namespace.

Exception: the use of anonymous namespaces in implementation files.

Recommendation 60

Do not use `using namespace`. It causes excessive namespace pollution that may lead to subtle errors. Always be explicit and use non-namespace `using`. This is also valid for the `std` namespace. Do not use `using namespace std;`.

Recommendation 61

Write namespace declarations in the following way:

- one namespace declaration per line
- the opening brace is on the same line as the declaration
- add a comment to the closing brace of the namespace

Example 15

Single-level namespace:

```
namespace Poco {


class ...
    ...
};


} // namespace Poco
```

Two-level namespace:

```
namespace Poco {
namespace XML {


class ...
    ...
};


} } // namespace Poco::XML
```

Recommendation 62

Prefer `using` over `typedef`.

C++11 allows to use the `using` keyword to define type aliases, which makes the code easier to read.

18 Error Handling

Rule 78

Use C++ exceptions instead of return values to report exceptional program states.

Rule 79

Always derive exception classes from `Poco::Exception` or one of its subclasses. Do not create a separate exception class hierarchy.

Rule 80

Destructors, deallocation and swap never fail (throw exceptions).

Recommendation 63

Use existing exception classes instead of defining new ones.

Recommendation 64

Always throw by value and catch by reference. Do not throw pointer types or types that are not subclasses of `std::exception`.

Recommendation 65

Do not use exception specifications (unless you are forced to).

Recommendation 66

Use assertions liberally to document internal assumptions and invariants. The POCO C++ Libraries provide useful macros for that purpose:

- `poco_assert()`
- `poco_assert_dbg()`
- `poco_check_ptr()`
- `poco_bugcheck()`

19 Portability

Rule 81

Never rely on implementation defined, unspecified or undefined behavior.

Examples for implementation defined behavior are:

- the number of bits in a byte,
- the size of an `int` or a `bool`,
- the signedness of `char`,
- the linkage of `main()`.

An example for unspecified behavior is the order in which function arguments are evaluated.

Examples for undefined behavior are:

- the effect of an attempt to modify a string literal,
- the effect of using an invalid pointer value,
- the result of a divide by zero.

Rule 82

Do not use platform-specific types in a public interface or in code that must be portable.

Examples for such types are `DWORD`, `TCHAR`, `LPTCSTR`, etc.

Rule 83

Do not rely upon absolute sizes of built-in types. If you need fixed-size types, use the ones provided by the POCO C++ Libraries (`Int8`, `Int16`, `Int32`, `Int64`, etc.). Do not assume that `int` and `long` have the same size, or that a pointer always fits into a `long`.

Rule 84

Take into account that different platforms have different byte orders and alignment requirements.

Rule 85

Clearly separate platform-specific from portable code.

20 References and Recommended Reading

The C++ Programming Language, 4th Edition

Bjarne Stroustrup

Addison-Wesley Professional, 2013

C++ Coding Standards

Herb Sutter and Andrei Alexandrescu

Addison-Wesley Professional, 2004

Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14

Scott Meyers

Addison-Wesley Professional, 2014

Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library

Scott Meyers

Addison-Wesley Professional, 2001

Large-Scale C++: Process and Architecture, Volume 1

John Lakos

Addison-Wesley Professional, 2019

Beautiful C++: 30 Core Guidelines for Writing Clean, Safe and Fast Code

J. Guy Davidson, Kate Gregory

Pearson Education, 2021

Designing and Building Portable Systems in C++

Günter Obiltschnig

Paper, Embedded Systems Conference Silicon Valley 2006

Programming in C++, Rules and Recommendations

Mats Henricson and Erik Nyquist

http://www.literateprogramming.com/ellemtel.pdf

The Joint Strike Fighter Air Vehicle C++ Coding Standards

Lockheed Martin Corporation

http://www.research.att.com/~bs/JSF-AV-rules.pdf

C++ Core Guidelines

Bjarne Stroustrup, Herb Sutter (Editors)

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

21 Appendix: Documentation

In order to keep project documentation and source code in sync, reference documentation is automatically generated from the source code. For this to work, the documentation for classes, functions, types, etc. has to follow certain conventions. These are described in the following.

21.1 General Conventions

Documentation that is to be included in the automatically generated reference documentation uses C++ single-line comments ("`//`"). To distinguish documentation from ordinary comments, a documentation comment begins with three slashes ("`///`").

The documentation is always written after the item to be documented. In the case of a class or function declaration/definition, the documentation is written immediately before the opening brace ("{"). Also for classes and functions, the documentation comment is indented one step deeper than the declaration.

A documentation line that begins with at least three whitespaces is formatted verbatim. This is useful for example code.

A documentation line that begins with at least three whitespaces, followed by either a minus sign or an asterisk, is formatted as a bullet list item.

A documentation line that begins with at least three whitespaces, followed by a digit and a period, is formatted as a numbered list item.

Paragraphs can be separated by inserting an empty documentation line.

21.2 Documenting Classes and Structs

The documentation for a class or struct is written after the "`class`" keyword, class name and inheritance list, but before the opening brace.

Example:

```
class Foundation_API Timer: protected Runnable
    /// This class implements a thread-based timer.
    ///
    /// A timer starts a thread that first waits for a given start
    /// interval.
    /// Once that interval expires, the timer callback is called
    /// repeatedly in the given periodic interval.
    /// If the interval is 0, the timer is only
    /// called once.
    /// The timer callback method can stop the timer by setting the
    /// timer's periodic interval to 0.
    ///
    /// The timer callback runs in its own thread, so multithreading
    /// issues (proper synchronization) have to be considered when
    /// writing the callback method.
```

```
    ///
    /// The exact interval at which the callback is called depends on
    /// many factors like operating system, CPU performance and system
    /// load and may differ from the specified interval.
    ///
    /// The timer thread is taken from the global default thread pool, so
    /// there is a limit to the number of available concurrent timers.
{
public:
    ...
};
```

21.3 Documenting Functions

The documentation for a function is written after the function declaration, but before the (if present) function body.

The parameters of a function are not documented separately. However, if a parameter requires special attention, a brief sentence or paragraph about it is written in the function documentation.

If the function may throw an exception, a paragraph should be written that describes which exceptions may be thrown under what circumstances.

If the function returns a value, the value should be described in a short paragraph.

Example 1:

```
void start(const AbstractTimerCallback& method);
    /// Starts the timer.
    /// Create the TimerCallback as follows:
    ///     TimerCallback<MyClass> callback(*this, &MyClass::onTimer);
    ///     timer.start(callback);
```

Example 2:

```
const std::string& operator [] (int index) const;
    /// Returns the index'th token.
    /// Throws a RangeException if the index is out of range.
```

Example 3:

```
    void unloadLibrary(const std::string& path)
        /// Unloads the given library.
        ///
        /// Be extremely cautious when unloading shared libraries.
        /// If objects from the library are still referenced somewhere,
        /// a total crash is very likely.
        ///
        /// If the library exports a function "pocoUninitializeLibrary",
        /// this function is executed before it is unloaded.
        ///
        /// If loadLibrary() has been called multiple times for the same
        /// library, the number of calls to unloadLibrary() must be the
        /// same for the library to become unloaded.
    {
        FastMutex::ScopedLock lock(_mutex);
```

```
        typename LibraryMap::iterator it = _map.find(path);
        ...
    }
```

21.4 Documenting Enumerations

The documentation for an enumeration is written after the `enum` keyword and the optional name, but before the opening brace. The documentation for each enumeration value is written directly after each value.

Example:

```
enum Options
    /// Flags that modify the matching behavior.
{
    GLOB_DEFAULT     = 0x00, /// default behavior
    GLOB_DOT_SPECIAL = 0x01, /// '*' and '?' do not match '.' at
                             /// beginning of subject
    GLOB_DIRS_ONLY   = 0x80  /// only glob for directories
};
```

21.5 Documenting Types

Documentation for at type definition is written immediately after the `typedef` statement.

Example:

```
typedef Int64 TimeVal; /// UTC time value in microsecond resolution
```

21.6 Libraries, Packages and Modules

Every source file that is part of a library must also belong to a package and a module. Packages and modules are used to further group functionality in a library. A package is a group of related classes. Examples are all classes belonging to the SAX (Simple API for XML) in the XML library, or all stream classes in the Foundation library. A module usually contains a header file and its accompanying implementation file, as well as any other files that cannot be removed without breaking the module.

Libraries and packages are used to structure the automatically generated reference documentation. Packages are used by the PocoBuilder application to enable the selection of a subset of the packages and modules in a library to be included in a custom build of that library.

See Example 1 for how to put library, package and module information in a source file.

22 Appendix: Abbreviations

Following is a list of common abbreviations that can be used in names.

Impl	Implementation
Param	Parameter
Ptr	Pointer
Prop	Property
Val	Value